Ejercicio de usabilidad de Ghosts

Ricardo Jacas

16 de diciembre de 2013

Problema 1, Cuentas Internacionales

Se desea hacer un programa que represente la interacción de un usuario con su cuenta bancaria, en cajeros automáticos que manejan distintos tipos de moneda.

Para ello debe crear las clases `User` y `Account` que representan al usuario y a su cuenta, respectivamente. Además se cree la clase `ATM` que representa a los cajeros automáticos. Por simplicidad, el sistema sólo debe manejar Dólares y Pesos, mediante las clases `Dollar` y `Peso`, ambas clases deberán proveer el método `getQuantity` que retorna la cantidad, en `double`, de dinero que representan.

Cada `User` podrá retirar dinero, en algún tipo de moneda, mediante su método `getMoney`, que extraerá dicho dinero de un `ATM`, indicando también de que cuenta (`Account`) y cuanto quiere retirar, expresado como `double`. Si la cantidad solicitada supera la cantidad presente en la cuenta, esta debe retornar el máximo presente en la cuenta.

Las cuentas se crearán para un tipo de moneda específica y un usuario en particular. La cantidad de dinero que almacenan debe ser representada por un tipo de moneda, no por un número. Estas deberán proveer el método `getAmount` que retorna la cantidad de dinero, en pesos/dólares que queda en la cuenta.

Los `ATM` también deben estar asociados a un tipo de moneda en específico, pero deben permitir la extracción de dinero de cualquier cuenta. Lo anterior mediante el método `getMoney`, que dado un usuario y una cuenta específicos, junto con una cantidad de dinero representada por un número (en `double`), retornará el dinero solicitado, en la moneda que maneja. Además los `ATM` deberán proveer las operaciones:

- `depositMoney` que dado un usuario, una cuenta y una cantidad de dinero, representada por un número (en `double`), aumenta la cantidad de dinero en la cuenta, guardando cuidado con la moneda que maneja el `ATM`.
- `moveMoney` que dado un usuario, una cuenta propia, una cuenta ajena y una cantidad de dinero representada por un número (en `double`), debe traspasar el dinero de una cuenta a otra, teniendo en consideración los tipos de monedas que manejan ambas cuentas, y por el `ATM`. Además deberá retornar `true` si la operación es exitosa y `false` si no lo es.

No debe crear jerarquías de `ATM` o de `Account`, sólo preocúpese que ambas clases manejen los dos tipos de moneda.

Para sus pruebas, cree la clase `TestAccounts` que extienda de `TestCase` y escriba los siguientes métodos de prueba:

- `testGetMoney`: Para probar el retiro de dinero por parte de un usuario, de su cuenta, en un cajero que utiliza la misma moneda que su cuenta.
- `testGetTonsOfMoney`: Para probar el retiro de dinero por parte de un usuario, de su cuenta, en un cajero que utiliza la misma moneda que su cuenta, pero pidiendo más dinero del que su cuenta contiene.
- `testGetDollarFromPeso`: Para probar el retiro de dinero por parte de un usuario, de su cuenta en pesos, en un cajero que utiliza dólares.
- `testGetPesoFromDollar`: Para probar el retiro de dinero por parte de un usuario, de su cuenta en dólares, en un cajero que utiliza pesos.
- `testDepositDollarFromPeso`: Para probar el depósito de dinero por parte de un usuario, de su cuenta en pesos, en un cajero que utiliza dólares.
- `testDepositPesoFromDollar`: Para probar el depósito de dinero por parte de un usuario, de su cuenta en dólares, en un cajero que utiliza pesos.

- `testMoveFromPesoToDollarInPeso`: Para probar el movimiento de dinero por parte de un usuario, de su cuenta en pesos a una cuenta en dólares, en un cajero que utiliza pesos.

Pregunta 2, Sistema de archivos

Un sistema de archivos cotidiano está compuesto por archivos y capetas, que contienen archivos y/o carpetas.

Se desea implementar un sistema de archivos donde cada carpeta tenga ordenados, por nombre, sus elementos. Para ello se implementarán las clases `NFolder` y `NFile`. Por simplicidad ambas clases se representarán por un nombre y tendrán un método `getPath` que retornará un objeto de tipo `NPath` para representar su ruta en el sistema de archivos. Además, las carpetas tendrán los siguientes métodos:

- `getSize`: para obtener la cantidad de archivos (`NFile`) que posee.
- `getElem`: que dada una posición, retorna el elemento en dicha posición.
- `addElem`: para agregar un elemento.
- `removeElem`: para quitar un elemento.
- `findFile`: que busca un archivo, recursivamente, por su nombre, y lo retorna. Si no existe, retorna `null`.
- `moveElem`: para mover un elemento a otra carpeta.

Los `NPath` deberán implementar 2 métodos: `getFullPath` y `getRelativePath`, que retornan la ruta completa, como `String`, del elemento a la carpeta más arriba en el sistema de archivos y la ruta relativa del elemento a una carpeta específica, respectivamente. Para la ruta relativa, utilice la notación clásica, esto es, describa la ruta de la forma `./carpeta1/carpeta2/archivo` donde el punto representa la carpeta de la cual se pide la ruta relativa.

Para reforzar que nuestras carpetas sólo puedan contener archivos y carpetas, y que estos estén ordenados por nombre, cada carpeta contendrá internamente una lista, representada por la clase `NList` que se encargará de contener los archivos dentro de la carpeta. **Tenga presente que `NList` no es una clase generica, solamente maneja `NFolder` y `NFile`.**

Para implementar `NList` cree una lista doblemente enlazada utilizando una clase `NNode` que se encargue de manejar los conceptos de sucesor y antecesor, y contener un elemento. Usando lo anterior implemente `NList` con los siguientes métodos:

- `size`: para obtener su cantidad de elementos.
- `get`: que dada una posición, retorna el nodo en dicha posición.
- `add`: para agregar un nodo.
- `remove`: para quitar un nodo.

Para sus pruebas, cree las clases `TestFileSystem` y `TestList` que extienda de `TestCase` y escriba los siguientes métodos de prueba:

Para `TestList`

- `testAdd`: Para probar que se generan los cambios esperados al agregar un nodo.
- `testAddRemove`: Para probar que la lista se mantiene coherente al agregar y quitar un elemento.
- `testAddOrdered`: Para probar si efectivamente los elementos están ordenados dentro de la lista.

Para `TestFileSystem`

- `testAddFile`: Para probar que se generan los cambios esperados al agregar un archivo.
- `testAddFileOrdered`: Para probar si efectivamente los archivos están ordenados dentro de la lista.
- `testAddFolder`: Para probar si al agregar una carpeta a otra carpeta todo sigue siendo coherente, esto es, que los métodos `size` y `getPath` funcionan correctamente.
- `testFind`: Para ver si la búsqueda de un archivo se realiza correctamente.
- `testPath`: Para probar si la operación `getPath` funciona correctamente al introducir archivos a carpetas y al anidar carpetas, probando las operaciones de la clase `NPath`.
- `testMove`: Para ver si al mover un elemento este desaparece del lugar de origen y aparece en el de destino. Recuerde probar si los `path` se actualizan.